Typed Regions *

Stefan Monnier and Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520-8285
{monnier, shao}@cs.yale.edu

Abstract. Standard type systems are not sufficiently expressive when applied to low-level memory-management code. Such code often uses some form of strong update (i.e. assignments that change the type of the affected location) and needs to reason about the relative position of objects in memory. We present a novel type system which, like alias types [20], provides a form of strong update, but with the advantage that it does not require the aliasing pattern to be statically described. It also provides operations over sequential memory locations and allows covariant reference casts. We then show how this new type system can be used to implement a type-safe stop&copy garbage collector that can properly collect cyclic data-structures. More specifically, we show how to write a two-generations collector for a language with mutable ref cells.

1 Introduction

As the technology of certifying compilation and proof carrying code [13, 1, 6] has progressed, the need to ensure the safety of the runtime system has increased. After all, if you go through the trouble of writing a foundational proof of safety of your code, you would rather not trust an unverified conservative garbage collector (GC) with your data. For this reason, it is very important to be able to write a type-safe GC, but the state of the art in this matter is still completely impractical since they cannot even handle cyclic data-structures. This paper's main goals are thus:

- Show that, in order to type-check a GC that can collect cyclic data-structures, the type system has to provide a form of strong update that can change the type of a location even if the set of aliases to this location is completely unknown.
- Present a type system that provides such a facility. This type system allows the programmer to choose any mix of linear or intuitionistic typing of references and to change this choice over time to adapt it to the current needs.
- Implement the first type-preserving GC that properly collects cycles and the first generational GC that allows the mutator to use destructive assignment.

* This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9901011 and CCR-0081590. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Traditional type systems are not well-suited to reason about type-safety of low-level memory management such as explicit memory allocation, initialization, deallocation, or reuse. Existing solutions to these problems either have a very limited applicability or rely on some form of *linearity* constraint. Such constraints tend to be inconvenient and a lot of work has gone into relaxing them. For example, the alias types system [20] is able to cleanly handle several of the points above, even in the presence of arbitrary aliasing, as long as the aliases can be statically tracked by the type system.

The reason why it is challenging to show type-safety of low-level memory management is that for this kind of code, the line between *type-safety* and *correctness* is blurred: we end up having to prove some non-trivial properties about the code just to show its type-safety. For example, type-safety of a generational GC depends on the correct processing of the remembered-set (a data-structure holding the set of pointers from the old generation to the new).

An alternative approach would be to use Hoare logic [8] to show the correctness of the low-level code and then provide a type-safe interface to it. But it is not clear how the two would interact, especially when the low-level code might be spread over a lot of code, as is the case for the code that maintains the remembered-set in the mutator. Furthermore as we start to encode more properties into our type systems than basic type-safety, the difficulties we are seeing here will start to appear for more mundane code as well. This tendency can already be seen in the Vault project which uses an approach taken from alias-types to prove other properties of their code than just type-safety.

The present work is thus an attempt to provide a middle ground between Hoare logic and simple type-systems. Additionally to the above stated goals, we make the following contributions:

- A language that can seamlessly combine the benefits of traditional *intuitionistic* references and *linear* references at the same time.
- We introduce type cast and strong update operations that work in the absence of any static aliasing information.
- The language also offers the ability to iterate over the objects contained in a sequential area of memory.
- We show how the technique described in [16] of using the calculus of inductive constructions (CiC) as our type language to track other properties than just type-safety can be used to track properties of state.

Section 2 introduces the problem of cyclic data-structures as well as two type systems on which our work is built. Section 3 presents the problems that have pushed us to develop this system. Section 4 describes our new language. Section 5 shows how we use it to write a generational GC. We then discuss related work a conclude.

2 Background

2.1 Cyclic structures

In the course of writing the *copy* routine of a garbage collector, we discovered that although current type systems can handle the case where the graph is acyclic, generalizing

the code to properly handle cycles proves difficult. After experimenting with various algorithms, it became clear that the problem is more fundamental: current type systems are unable to type-check generic code that can build arbitrary cyclic data-structures.

To see this, let us look at a classic example, a datatype for doubly-linked lists:

$$\mathsf{datatype}\ \alpha\ \mathit{dlist} = \mathit{Node}\ \mathsf{of}\ \alpha * \alpha\ \mathit{dlist\ ref} * \alpha\ \mathit{dlist\ ref}$$

The SML type system allows us to declare this datatype and write functions to manipulate it, but does not offer us any way to create such an object: we need a base case. So let us take another example, with a base case:

$$\mathsf{datatype}\ \alpha\ \mathit{tree} = \mathit{Node}\ \mathsf{of}\ \alpha \mid \mathit{Branch}\ \mathsf{of}\ \alpha\ \mathit{tree\ ref} * \alpha\ \mathit{tree\ ref}$$

This time, we can construct such trees since we do have a base case, but only if we have an object of the proper type α. More specifically, in order to create a cyclic data-structure, we always need a base case to start from, even if the data-structure we want to get in the end does not contain any node of this base case any more. Which means that a generic routine such as an unpickler or a copying GC needs to be able to construct from scratch the base case of any type that could be involved in a cyclic data-structure. In the tree example above, that means creating a *Node* $\mathsf{of}\ \alpha$ for any α. Clearly this is not possible.

OCaml provides special support to build cyclic data-structures such as the *dlist* example above using a construct similar to $\mathsf{val\ rec}\ n = \mathit{Node}(0, n, n)$. This helps for specific code, but is of no use for generic code since it only works for pre-determined cycles, whereas a copying GC simply does not even know when it is creating cycles.

Type systems that can decouple allocation from initialization are key to solving this problem, but none of the systems developed so far are sufficiently flexible to handle the case of a generic function such as *unpickle*. More specifically, none of them know how to handle the case where the pointer to the allocated object *escapes* (i.e. is passed around and stored at arbitrary locations) before the object is initialized: when we allocate a new object, we obviously know its one and only alias, but we cannot initialize it because some of the values might not exist yet, and by the time we are done unpickling the children such that initialization can take place, there can be any number of aliases and we do not statically know them because the function is generic.

In order to type-check a practical copying GC, we need a new type system that is able to update the type (e.g. from uninitialized to initialized) of all the aliases to a particular object even when those aliases are not statically known.

2.2 Regions

Region-based type systems [18, 3] are the most practical systems offering type-safe explicit memory management. They provide a solution to the problem of safe deallocation, with a minimum of added constraints. Even though they do not offer any help when trying to type-check low-level code such as object initialization, their practicality makes them very attractive as a starting point. The idea behind region calculi is to manage memory at the level of regions (groups of objects), to annotate the type of every

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa ::= \Omega \mid \mathsf{R} & (\textit{heap type}) \;\; \Psi ::= \bullet \mid \Psi, \nu.n \mapsto \sigma \\
(\textit{regions}) & \rho ::= r \mid \nu & (\textit{type env}) \;\; \Gamma ::= \bullet \mid \Gamma, x{:}\sigma \\
(\textit{types}) & \sigma ::= t \mid \mathsf{int} \mid \sigma \times \sigma \mid \sigma \;\mathsf{at}\; \rho & (\textit{kind env}) \;\; \Delta ::= \bullet \mid \Delta, t{:}\kappa \\
 & \quad \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 & (\textit{region env}) \; \Theta ::= \bullet \mid \Theta, \rho
\end{array}
$$

$$
\begin{array}{ll}
(\textit{values}) & v ::= x \mid i \mid (v, v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & op ::= v \mid \pi_i v \mid \mathsf{put}[\rho]\; v \mid \mathsf{get}\; v \\
(\textit{terms}) & e ::= v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\; v \mid \mathsf{let}\; x = op\; \mathsf{in}\; e \mid \mathsf{set}\; v := v; e \\
 & \quad \mid \mathsf{let}\; r = \mathsf{newrgn}\;\; \mathsf{in}\; e \mid \mathsf{freergn}\; \rho; e
\end{array}
$$

Fig. 1. Syntax of a region-based language.

pointer with the region that it references, and to only provide bulk deallocation of a whole region at a time.

Figure 1 shows how such a language might look like: $\mathsf{put}[\rho]\; v$ allocates v in region ρ; $\mathsf{get}\; v$ dereferences v; newrgn creates a new region and $\mathsf{freergn}$ deallocates it; $\nu.n$ is a pointer to the n^{th} object in region ν created by put and has type $\sigma\;\mathsf{at}\;\nu$ if the object referenced has type σ (i.e. $\Psi(\nu.n) = \sigma$). Functions have type $\forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0$; they are fully closed and we use continuation passing style, so they never return (hence the $\to 0$ in the type); Δ is the list of type (and region) parameters; Θ lists the regions that need to be live at the time of the call; and $\vec{\sigma}$ lists the type of the value parameters. A function call $v[\vec{\sigma}](\vec{v})$ passes types $\vec{\sigma}$ and values $\vec{v}$ to function v. [1]

Here is a sample function that creates a cyclic node of the *tree* datatype presented previously, assuming the language has been extended with support for datatypes:

$$
\begin{array}{l}
\textbf{\textit{mktree}}[r,t]\{r\}(x{:}t, k : \forall[r,t]\{r\}((\textit{tree}\; t)\;\mathsf{at}\; r) \to 0) \\
= \mathsf{let}\; n = \mathsf{put}[r]\; (\textit{Node}\; x)\; \mathsf{in} \\
\quad \mathsf{set}\; n := \textit{Branch}\; n\; n; k[r,t](n)
\end{array}
$$

The function is parameterized over type t and region r and expects an argument x of type t (which is only used temporarily to create the dummy *Node*) and a continuation argument k. The put operation does the allocation while the set operation does the initialization. The (omitted) kind of r is R and the kind of t is Ω. If k's type had $\{\}$ in place of $\{r\}$, it would force us to deallocate the region r before calling it and it would make n into a dangling pointer, which is allowed because liveness of the region is only needed and checked when dereferencing with get.

2.3 Alias types

The alias-types system [17, 20] was developed precisely to handle low-level code such as object initialization, memory reuse, and safe deallocation at the object level. To do that, the type of pointers is changed to carry no information about the type of the referenced object. Instead, the type of a pointer is just the location it is pointing to, so it does

[1] Region calculi are not necessarily using fully closed functions and continuation passing style like we do here, but a direct-style presentation is more complex, as is the correct treatment of closure allocation.

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa ::= \Omega \mid \mathsf{Heap} \mid \mathsf{Loc} & \\
(\textit{locations}) & \rho ::= r \mid \nu & \\
(\textit{types}) & \sigma ::= t \mid \mathsf{int} \mid \nu & \\
 & \quad \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 & \\
 & & \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x{:}\sigma & \\
(\textit{kind env}) & \Delta ::= \bullet \mid \Delta, t{:}\kappa & \\
(\textit{mem env}) & \Theta ::= \bullet \mid \Theta, \rho \mapsto (\sigma, ..., \sigma) &
\end{array}
$$

$$
\begin{array}{ll}
(\textit{values}) & v ::= x \mid i \mid \nu \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & op ::= v \mid \pi_i v \\
(\textit{terms}) & e ::= v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{set}\ \pi_i v := v; e \\
 & \quad \mid \mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e \mid \mathsf{free}\ \rho; e
\end{array}
$$

Fig. 2. Syntax of an alias-types language.

not need to change when the location's type or liveness changes. While it provides a lot of power when dealing with low-level code, it relies on an amount of static information which is rarely available in general and definitely not available in our copying GC.

Figure 2 shows the syntax of a very simple alias-types language. It can be thought of as a region-based language where the pointers can only point to regions rather than to objects inside them and where regions have been turned into tuples. put and get have disappeared and the environment Ψ mapping locations to their types has been merged into Θ. When dereferencing a pointer of type ρ, we thus have to check the liveness and the type of the corresponding location by looking up ρ in Θ. $\mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e$ allocates a new object of size n and returns the location as both a value x and a type r. We could also have done this for regions so as to distinguish between the region type and the region value passed to put at runtime, but we conflated the two for simplicity.

Here is a sample code that takes a value of type t and creates an infinite list of this element (a 1-element circular list):

$$
\begin{array}{l}
\textbf{\textit{mklist}}[\epsilon, t]\{\epsilon\}(x : t, k{:}\forall[\epsilon, t, r]\{\epsilon, r \mapsto (t, r)\}(r) \to 0) \\
= \mathsf{let}\ (r, n) = \mathsf{new}\ 2\ \mathsf{in} \\
\quad \mathsf{set}\ \pi_0 n := x; \mathsf{set}\ \pi_1 n := n; k[\epsilon, t, r](n)
\end{array}
$$

The argument ϵ has kind Heap and means that the function accepts an arbitrary heap as input, whereas the type of the continuation k shows that the returned heap is ϵ extended with a circular node at location r. Since new only knows about the size of the object, it can only do allocation and the type at location r is originally set to $(\mathsf{int}, \mathsf{int})$ and is then incrementally updated by each set operation to (t, int) and then (t, r).

The ability to update a location's type is the key power of alias-types. But for that it relies crucially on the fact that the type system should be able to keep track of pointer values. In particular, the types need to statically but precisely describe the shape of the heap. Witness the fact in the above example that the type of the circular list is not just *list* t but instead explicitly describes a 1-element cycle and thus disallows any other shape. The type language of [20] is much richer than what we show here, but that does not help when the shape of the heap is simply not known statically.

2.4 Calculus of constructions

The calculus of inductive constructions (CiC) [14] that we use as our type language is an extension of the calculus of constructions (CC) [2], which is a higher-order typed logic. CC can encode Church's higher-order predicate logic via the Curry-Howard isomorphism [9]. Understanding the details of this language is not necessary for this paper. Suffice it to say that it is a powerful typed λ-calculus where λ-terms have types of the form $\Pi x : \varphi.\, \varphi$ which subsume both the usual arrow and the universal quantifier. Its inductive definitions are like datatypes, with elimination constructs which combine case analysis with a fixpoint operation. To know a bit more about CiC, see the appendix A.

3 Motivating example

The simple type-preserving GCs developed until now have a few important limitations. The most important one is that cyclic garbage is not properly collected. Another limitation is that generational collectors do not allow the mutator (i.e. the code that uses the GC) to create references from the old region to the young via destructive assignment. We want to lift those two restrictions, since they make those type-preserving GCs impractical.

3.1 Simple type-preserving GC

The basic idea of a type-preserving GC, proposed by Wang and Appel [22], is to layer a stop&copy collector on top of a region calculus, where the whole heap is placed in a single region and where the *copy* function copies the heap from the *from* region to a new *to* region and then frees the *from* region.

Because the type of the heap contains region annotations which will necessarily be different before copying than after, the *copy* function cannot just be of type $t \to t$ but instead has to be of the form $M_F(t) \to M_T(t)$ where t represents what should be preserved while the M type function annotates it with details that the mutator does not care about. Furthermore, in order to work correctly, the *copy* routine might require things like tag bits, mark bits or in case of generational GC it will need to make sure the mutator obeyed the generation barriers and provides a correct remembered set. All those added constraints will need to be somehow encoded in M since *copy* has obviously no control over t.

A good way to look at it is that t is a high-level, GC-oblivious type of the heap, while $M_F(t)$ is the low-level representation with all the added details necessary for the correct functioning of the GC. The two will generally not be of the same kind; t might for example correspond to the type used in the high-level source language of the mutator code.

The simplest case is a bare-bones stop&copy GC with a source language that only supports integers and pairs. M might then look like the following, where R is the current region holding the heap:

$$\begin{aligned} M_R(\mathsf{int}) &\Rightarrow \mathsf{int} \\ M_R(\tau_1 \times \tau_2) &\Rightarrow (M_R(\tau_1) \times M_R(\tau_2)) \text{ at } R \end{aligned}$$

In the case of a 2-generation collector, the heap will be spread over 2 regions: Y for the nursery and O for the old generation. M will now have to enforce that there cannot be any pointer back from O to Y by only allowing $(M_{O,Y}(\tau_1) \times M_{O,Y}(\tau_2))$ at Y and $(M_{O,O}(\tau_1) \times M_{O,O}(\tau_2))$ at O:

$$\begin{array}{ll} M_{O,A}(\mathsf{int}) & \Rightarrow \mathsf{int} \\ M_{O,A}(\tau_1 \times \tau_2) & \Rightarrow \exists r \in \{O, A\}.(M_{O,r}(\tau_1) \times M_{O,r}(\tau_2)) \text{ at } r \end{array}$$

In this definition, A is expected to be either equal to O or Y depending on whether pointers to Y are allowed at this place.

3.2 Generational collection

We want to create a simple yet usable type-preserving generational GC. In order to be usable, it needs to be able to properly handle cycles in the heap and it needs to allow the mutator to use destructive assignment, even if that means creating pointers back from the old generation to the nursery. The traditional way to deal with such pointers is to keep track of all of them in a *remembered-set*, but for simplicity, we will restrict mutability to ref-cells, as is done in SML, and place all those ref-cells in a separate region which will thus play the role of a degenerate remembered-set.

Compared to the situations outlined in the previous section, the heap is now spread over three regions: Y for the nursery, O for the old generation and R for the ref-cells. So that the collection of Y can take place without having to scan O, we need to make sure that there is no pointer from O to Y. I.e. pointers from Y and R can point to any of Y, O or R but pointers from O can only point to O or R. Our M type function which maps high-level types to their low-level representation will again have to enforce this constraint. If our high-level types include integers, ref-cells and pairs, M could be defined as follows:

$$\begin{array}{ll} M_{Y,R,O,A}(\mathsf{int}) & \Rightarrow \mathsf{int} \\ M_{Y,R,O,A}(\mathsf{ref}\ \tau) & \Rightarrow M_{Y,R,O,Y}(\tau) \text{ at } R \\ M_{Y,R,O,A}(\tau_1 \times \tau_2) & \Rightarrow \exists r \in \{O, A\}.(M_{Y,R,O,r}(\tau_1) \times M_{Y,R,O,r}(\tau_2)) \text{ at } r \end{array}$$

Here A is expected to be either equal to O or to Y and $M_{Y,R,O,A}(\tau_1 \times \tau_2)$ is the type of a pair that can be allocated either in O or in A and whose children (and grand-children, etc...) might also refer additionally to R or to Y.

The GC code that copies objects from the nursery Y to the old space O takes the type of the heap t, the regions Y, O, and R, and the heap itself $h : M_{Y,R,O,Y}(t)$ as well as a continuation k. It first copies everything reachable from the root h (but only within Y) to O. Then scans all the ref-cells, considered as extra roots, and copies anything reachable from them as well. Finally, it frees Y and creates a new nursery before returning to the continuation k. It could look like the following:

$$\begin{array}{l} \boldsymbol{GC}[t : \Omega^\tau]\{Y, R, O\}(h : M_{Y,R,O,Y}(t), k : \dots) \\ = \mathsf{let}\ h = \mathit{copy}[t, Y, R, O](h)\ \mathsf{in} \\ \quad \mathsf{let}\ t', x = \mathsf{first}\ R\ \mathsf{in} \\ \quad \mathsf{let}\ \_ = \mathit{redirect}[t', Y, R, O](x)\ \mathsf{in} \\ \quad \mathsf{freergn}\ Y; \mathsf{let}\ Y = \mathsf{newrgn}\ \ \mathsf{in}\ k[Y, R, O](h) \end{array}$$

where the *copy* function should simply do a generic deep copy, but only within the bounds of Y and the *redirect* function should scan the R region, redirecting any pointer to the Y region to a copy of that object in O. The type of the new root pointer h returned by *copy* can be described as $M_{Y,R,O,O}(t)$.

In order to be able to free Y at the end of the collection, we need to keep track somewhere at the type-level of the fact that pointers in R that have been redirected can only point to O or to R but not to Y any more. Indeed, as the redirection is taking place the type of h keeps changing gradually until the end when it becomes equivalent to $M_{O,R,O,O}(t)$ and thus independent from Y. If b is the boundary between what has been redirected and what has not, then we can write $h : C_{Y,R,O,O,b}(t)$ where C is defined as:

$$
\begin{array}{ll}
C_{Y,R,O,A,b}(\mathsf{int}) & \Rightarrow \mathsf{int} \\
C_{Y,R,O,A,b}(\mathsf{ref}\ \tau) & \Rightarrow \mathsf{let}\ r = \mathsf{if}\ (\textit{before}\ b)\ O\ Y\ \mathsf{in}\ C_{Y,R,O,r,b}(\tau)\ \mathsf{at}\ R \\
C_{Y,R,O,A,b}(\tau_1 \times \tau_2) & \Rightarrow \exists r \in \{O, A\}.(C_{Y,R,O,r,b}(\tau_1) \times C_{Y,R,O,r,b}(\tau_2))\ \mathsf{at}\ r
\end{array}
$$

This is almost the same as the previous M, except for the added b parameter and the case of ref where the object reachable from the ref-cell will have either type $C_{Y,R,O,O,b}(\tau)$ or $C_{Y,R,O,Y,b}(\tau)$ depending on whether they have been redirected or not.

The above presentation is sloppy and simplistic but already requires unusual features from the language:

- The *before* b test in the definition of $C_{Y,R,O,A,b}(\mathsf{ref}\ \tau)$ obviously needs to know not just the region in which the object is located but its actual location, which implies that object locations need to be reflected in the pointer type.
- The new operation first (as well as ifnext which will be needed inside *redirect*), that allows the code to scan the region R, should return both a new pointer x and a new type t' where x is expected to be of type $C_{Y,R,O,Y,b}(t')$. But this is only safe if we know that all objects in the relevant section have a type of the form $C_{Y,R,O,Y,b}(\tau)$.
- Since the type of h, indexed by the boundary b keeps changing as the boundary moves, we need to be able to update its type when the assignment $x := \textit{copy}[..](..)$ moves the boundary, even though we do not know anything about the aliasing relationship between x and h.

The exact same three issues appear when trying to write a Cheney-style stop&copy GC where the *to* region is used as a queue and needs to be scanned and redirected in very similar ways.

4 Typed regions

Our new system of typed regions solves all of those problems. It can be thought of as a hybrid between alias-types and the calculus of capabilities [3] supplemented with the calculus of inductive constructions (CiC), similarly to λ_H [16]. Where alias-types rely on a *linear* map of live locations' types and the calculus of capabilities relies on a linear set of live regions, we rely on a linear map of regions' types.

In a typical region calculus, the type of the object reachable from a pointer (its target) is entirely given by the type of the pointer. In contrast, in the alias-types system,

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa & ::= \Omega \mid \mathsf{R}\ \kappa \mid \kappa \to \kappa \\
(\textit{regions}) & \rho & ::= r \mid \nu \\
(\textit{types}) & \sigma, \varphi, \tau & ::= t \mid \mathsf{int} \mid \sigma \times \sigma \mid \tau\ \mathsf{at}\ \rho.n \\
 & & \quad \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0
\end{array}
$$

$$
\begin{array}{lll}
(\textit{heap type}) & \Psi & ::= \bullet \mid \Psi, \nu.n \mapsto \tau \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x{:}\sigma \\
(\textit{kind env}) & \Delta & ::= \bullet \mid \Delta, t{:}\kappa \\
(\textit{region env}) & \Theta & ::= \bullet \mid \Theta, \rho \mapsto (\varphi, n)
\end{array}
$$

$$
\begin{array}{lll}
(\textit{values}) & v & ::= x \mid i \mid (v, v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & \textit{op} & ::= v \mid \pi_i v \mid \mathsf{put}[\rho, \tau]\ v \mid \mathsf{get}\ v \\
(\textit{terms}) & e & ::= v[\vec{\varphi}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \textit{op}\ \mathsf{in}\ e \mid \mathsf{set}\ v \overset{\varphi}{:=} v; e \\
 & & \quad \mid \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e
\end{array}
$$

Fig. 3. Syntax of the core of our language.

the type of the pointer does not provide any direct information about the type of the target; instead, the target's type is kept in a linearly managed *type map* indexed by the pointer's type, which is the singleton type holding the object's location. Our new type system mixes the two, such that the pointer's type holds both the location and some information (called the *intended type*) about the object to which it points, while the remaining information is kept in a map of regions' types. The type of a region is a function that maps an object's location and its intended type to its actual type.

4.1 Overview

The syntax of a trimmed down version of our language, shown in Fig. 3, looks like the simple region calculus presented before, except for the following differences:

- The region environment Θ now contains not only a list of live regions, but a map from live regions to their type φ and size n.
- Pointer types have the form τ at $\rho.n$ rather than just σ at ρ, where n is the offset inside the region. Such a pointer does not point to an object of type τ. Instead, we force an indirection through the regions' type such that the target's type is $\Theta(\rho)\ n\ \tau$. Think of τ as the *intended type* of the location while $\Theta(\rho)$ maps those intended types to their actual value at any particular time.
- The region kind now takes a parameter κ. If $\rho : \mathsf{R}\ \kappa$, then the region's type will be of kind $\mathsf{Nat} \to \kappa \to \Omega$ and the term τ in τ at $\rho.n$ will have to have kind κ.
- Just as before, a value $\nu.n$ has type $(\Psi(\nu.n))$ at $\nu.n$ but $\Psi(\nu.n)$ can now be of any kind rather than only Ω.
- put takes an additional parameter τ and returns a pointer of type τ at $\rho.n$ after checking that $v : \Theta(\rho)\ n\ \tau$.
- set is now a strong update: it changes the type of the location to φ which has kind $\kappa \to \Omega$. This type needs to be provided because there are many valid choices and they are not all equivalent.
- newrgn now takes a parameter φ which is the initial type of the region.

A region's type $\Theta(\rho)$ which ignores the location parameter n corresponds to a classical intuitionistic region system where aliases cannot be distinguished; pointer types

will then typically be of the form $\exists n.\tau$ at $\rho.n$ since static tracking of n is unnecessary. On the other hand, if the type ignores the τ parameter, then it enjoys properties more like those of linear systems such as alias-types: you have to statically keep track of the exact location n, you can abstract away the intended type by using $\exists t.t$ at $\rho.n$, and you can change the type of a location without any regard to its previous use. The strength of our system resides in the fact that we can choose at any point from a continuum of options between those two extremes.

As mentioned, this system solves the three problems we have encountered when trying to code our generational GC: The *before* b test can simply compare b to the offset n of the pointer since it is now carried in its type; Since all objects in a region have a type of the form $\Theta(\rho)\ n\ t$, setting the region's type to something like C makes sure that first indeed returns a pointer to an object of type $C(t)$; Finally, since Θ is managed linearly and since all memory accesses have to look up the target's type in Θ, we can globally update the type of objects when we move the boundary by updating the corresponding type in Θ.

4.2 The language

The syntax of the full language is shown in Fig. 4. The language uses continuation passing style and fully closed functions. *sort* and *ptm* together form the pure type system corresponding to CiC. All our types are actually terms of this language and Ω is actually defined as an inductive definition in CiC. $M, \Psi, \Gamma, \Delta, \Theta$ are environments used in the typing rules.

The region spec stored in the region environment Θ is either the type of the region together with its size (φ, n) or the set of regions for which this region can be an alias. This is used when it is not statically known into which region a reference will point, in which case, the reference will have type $\exists r \in \vec{\rho}.\tau$ at $r.n$.

Values can be integers, pairs ($(v, v) : \sigma \times \sigma$), references ($\nu.n : \tau$ at $\nu.n$), existential packages ($\langle \varphi, v \rangle : \exists t : \kappa.\sigma$), region existential packages ($\langle \rho, v \rangle : \exists r \in \vec{\rho}.\sigma$) and functions. The terms do the following:

$\pi_i v$: select from a tuple.
put$[\rho, \tau]\ v$: allocate an object v in region ρ.
get v: return the object pointed to by v.
cast$[P]\ v$: safely change the type of v according to proof P.
$v[\vec{\varphi}](\vec{v})$: make a tail-call to function v.
halt v: halt the machine.
let $x : \sigma = op$ in e: bind variable x as you would expect.
set $v :\overset{\varphi}{=} v; e$: do a strong update. φ describes the new type of the location.
let $r =$ newrgn φ in e: allocate a new region of type φ.
freergn $\rho; e$: free the region ρ.
cast$[P]\ \rho \mapsto \varphi; e$: safely change the type of region ρ to φ, with proof P.
let $\langle t, x \rangle =$ open v in e: unpack the existential v into t and x.
let $\langle r, x \rangle =$ openrgn v in e: unpack the region existential v into r and x.
iffirst $\rho\ (x, t\ .\ e)\ (e)$: test whether ρ is empty and selects the corresponding branch. If non-empty, x is bound to a pointer to the first object and t to its type.

$$
\begin{array}{lrll}
(\textit{sort}) & & s & ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{ptm}) & & \varphi, \tau, \kappa, P & ::= s \mid x \mid \lambda x{:}\varphi.\, \varphi \mid \varphi\ \varphi \mid \Pi x{:}\varphi.\, \varphi \\
 & & & \quad \mid \mathsf{Ind}(x{:}\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i, \varphi) \mid \mathsf{Elim}[\varphi, \varphi](\varphi)\{\vec{\varphi}\} \\
(\textit{memory}) & & M & ::= \bullet \mid M, \nu.n \mapsto v \\
(\textit{heap type}) & & \Psi & ::= \bullet \mid \Psi, \nu.n \mapsto \tau \\
(\textit{type env}) & & \Gamma & ::= \bullet \mid \Gamma, x{:}\sigma \\
(\textit{kind env}) & & \Delta & ::= \bullet \mid \Delta, t{:}\kappa \\
(\textit{region env}) & & \Theta & ::= \bullet \mid \Theta, \rho \mapsto \theta \\
(\textit{region spec}) & & \theta & ::= (\varphi, n) \mid \vec{\rho} \\
(\textit{regions}) & \rho{:}\mathsf{R}\ \kappa & & ::= r \mid \nu \\
(\textit{types}) & \sigma{:}\Omega & & ::= \mathsf{int} \mid \sigma \times \sigma \mid \tau\ \mathsf{at}\ \rho.n \mid \exists t{:}\kappa.\sigma \mid \exists r \in \vec{\rho}.\sigma \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
(\textit{values}) & & v & ::= x \mid i \mid (v, v) \mid \nu.n \mid \langle \varphi, v \rangle \mid \langle \rho, v \rangle \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & & op & ::= v \mid \pi_i v \mid \mathsf{put}[\rho, \tau]\ v \mid \mathsf{get}\ v \mid \mathsf{cast}[P]\ v \\
(\textit{terms}) & & e & ::= v[\vec{\varphi}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x{:}\sigma = op\ \mathsf{in}\ e \mid \mathsf{set}\ v \overset{\varphi}{:=} v; e \\
 & & & \quad \mid \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e \mid \mathsf{cast}[P]\ \rho \mapsto \varphi; e \\
 & & & \quad \mid \mathsf{let}\ \langle t, x \rangle = \mathsf{open}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ \langle r, x \rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e \\
 & & & \quad \mid \mathsf{iffirst}\ \rho\ (x, t\ .\ e)\ (e) \mid \mathsf{ifnext}\ v{+}i\ (x, t\ .\ e)\ (e)
\end{array}
$$

Fig. 4. Syntax of the language

ifnext $v{+}i\ (x, t\ .\ e)\ (e)$: similarly, give a pointer to the next object past v, if any. i is the size of the object pointed to by v and thus skipped by the operation.

We have decided to manipulate whole objects rather than words and not to split allocation from initialization. It is easy to change the language to provide alloc, load, and store instead of put, get, and set, but the typing rules become more verbose and so does the code in our examples.

The operational semantics of the language are shown in Fig. 5. The machine state is define as the 4-tuple $(M; \Theta; \Psi; e)$ where Θ is only used when checking the size of a region in iffirst and ifnext, and Ψ is only used to provide the type of the pointer in those same two operations. Instead of checking the size in Θ, we could look up $M(\nu.n)$ to see if it exists, which would get us rid of Θ, but that would move us further away from a realistic implementation.

The typing rules are given in the appendix, together with statements of type soundness and complete collection properties.

5 Generational collection

As suggested earlier, we will use a simple form of generational collection, so as to sidestep the issue of keeping track of the back references from the old generation to the nursery. So our source language is restricted to the following types: integers, immutable pairs, reference cells.

$$
\begin{array}{ll}
(M;\Theta;\Psi;\nu.n[\overrightarrow{\varphi}](\overrightarrow{v})) & \\
\quad \text{where } M(\nu.n) = \lambda[\overrightarrow{t:\kappa}]\{\Theta'\}(\overrightarrow{x:\sigma}).e & \Longrightarrow (M;\Theta;\Psi;e[\vec{\varphi},\vec{v}/\vec{t},\vec{x}]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = v\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[v/x]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = \pi_i(v_1,v_2)\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[v_i/x]) \\
(M;\Theta,\nu\mapsto(\varphi,n);\Psi; & \Longrightarrow (M,\nu.n\mapsto v;\Theta,\nu\mapsto(\varphi,n{+}1); \\
\quad \mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e) & \qquad \Psi,\nu.n\mapsto\tau;e[\nu.n/x]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[M(\nu.n)/x]) \\
(M;\Theta;\Psi;\mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[v/x]) \\
(M;\Theta;\Psi;\mathsf{let}\ \langle t,x\rangle = \mathsf{open}\ \langle\varphi,v\rangle\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[\varphi,v/t,x]) \\
(M;\Theta;\Psi;\mathsf{let}\ \langle r,x\rangle = \mathsf{openrgn}\ \langle\nu,v\rangle\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;\Psi;e[\nu,v/r,x]) \\
(M;\Theta;\Psi;\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta,\nu\mapsto(\varphi,0);\Psi;e[\nu/r]) \\
 & \quad \text{where } \nu\notin Dom(\Theta) \\
(M;\Theta;\Psi;\mathsf{freergn}\ \nu;e) & \Longrightarrow (M\backslash\nu;\Theta\backslash\nu;\Psi\backslash\nu;e) \\
(M;\Theta,\nu\mapsto(\varphi',n);\Psi;\mathsf{cast}[P]\ \nu\mapsto\varphi;e) & \Longrightarrow (M;\Theta,\nu\mapsto(\varphi,n);\Psi;e) \\
(M;\Theta,\nu\mapsto(\varphi',n');\Psi;\mathsf{set}\ \nu.n :\overset{\varphi}{=} v;e) & \Longrightarrow (M,\nu.n\mapsto v;\Theta,\nu\mapsto(\mathit{upd}\ \varphi'\ n\ \varphi,n');\Psi;e) \\
(M;\Theta;\Psi; & \Longrightarrow \begin{cases} (M;\Theta;\Psi;e_2) & \text{if } \Theta(\nu)=(\varphi,0) \\ (M;\Theta;\Psi;e_1[\tau,\nu.0/t,x]) & \text{where } \Psi(\nu.0)=\tau \end{cases} \\
\quad \mathsf{iffirst}\ \nu\ (x,t\ .\ e_1)\ (e_2)) & \\
(M;\Theta;\Psi; & \Longrightarrow \begin{cases} (M;\Theta;\Psi;e_2) & \text{if } \Theta(\nu)=(\varphi,n{+}1) \\ (M;\Theta;\Psi;e_1[\tau,\nu.n{+}1/t,x]) & \text{where } \Psi(\nu.n{+}1)=\tau \end{cases} \\
\quad \mathsf{ifnext}\ \nu.n{+}i\ (x,t\ .\ e_1)\ (e_2)) &
\end{array}
$$

Fig. 5. Operational semantics of the language.

We will use three regions: O is the old generation, Y is the nursery and R contains all the ref-cells and only ref-cells. Thus R can act as a degenerate form of remembered set. We require that pointers from O can only point to O or R but not to Y.

The type of our variables have the shape $M_{O,R,Y}(\tau)$ defined below:

$$
\begin{array}{ll}
M_{O,R,A}(\mathsf{int}) & \Rightarrow \mathsf{int} \\
M_{O,R,A}(\mathsf{ref}\ \tau) & \Rightarrow \exists n{:}\mathsf{Nat}.\tau\ \mathsf{at}\ R.n \\
M_{O,R,A}(\tau_1\times\tau_2) & \Rightarrow \exists r\in\{O,A\}.\exists n{:}\mathsf{Nat}.(\tau_1,\tau_2)\ \mathsf{at}\ r.n
\end{array}
$$

The regions' types while the mutator is running look like:

$$
\begin{array}{l}
Y\mapsto\lambda n.\lambda(t_1,t_2).M_{O,R,Y}(t_1)\times M_{O,R,Y}(t_2) \\
O\mapsto\lambda n.\lambda(t_1,t_2).M_{O,R,O}(t_1)\times M_{O,R,O}(t_2) \\
R\mapsto\lambda n.\lambda t.M_{O,R,Y}(t)
\end{array}
$$

and while a collection copying objects from Y to O is in progress and pointers in R are redirected from Y to O, the type of R is the following:

$$R\mapsto\lambda n.\lambda t.\mathsf{let}\ r = \mathsf{if}\ m > n\ \mathsf{then}\ O\ \mathsf{else}\ Y\ \mathsf{in}\ M_{O,R,r}(t)$$

The top-level code of the GC code that copies objects from the nursery Y to the old space O is shown in figure 6. We omit type parameters that are obvious from context

```
(* Inductive definition of the source-level types. *)
```
Inductive Ω^τ : Kind $:=$ *int* : Ω^τ | *ref* : $\Omega^\tau \to \Omega^\tau$ | *pair* : $\Omega^\tau \to \Omega^\tau \to \Omega^\tau$

```
(* Mapping types from source-level to low-level. *)
```
typedef $\boldsymbol{M}_{o,r,a}\ t =$ case t of *int* $\Rightarrow$ int
$\quad$ | *ref* t $\Rightarrow \exists n$:Nat.t at $r.n$
$\quad$ | *pair* $t_1\ t_2$ $\Rightarrow \exists b \in \{o,a\}.\exists n$:Nat.$(t_1,t_2)$ at $b.n$

```
(* Type of a region holding tuples, like o and y. *)
```
typedef ***TupRgn*** $o\ r\ a = \lambda n.\lambda(t_1,t_2).M_{o,r,a}(t_1) \times M_{o,r,a}(t_2)$

```
(* Type of a region holding ref cells, during redirection. *)
```
typedef ***RefRgn*** $o\ r\ y\ b = \lambda n.\lambda t.$let $a =$ if $(b > n)\ o\ y$ in $M_{o,r,a}(t)$

```
(* Type of the continuation passed to the GC. *)
```
typedef ***gccont*** $= \forall[y$:R $(\Omega^\tau \times \Omega^\tau)]\{\ldots\}(x : M_{o,r,y}(t)) \to 0$

```
(* A deep copy function not shown here. *)
```
copy : $\forall[t:\Omega^\tau, b$:Nat,$\ldots]\{\ldots\}(M_{o,r,y}(t)) \to M_{o,r,o}(t)$

```
(* Main entry point. h is the root pointer and t is its type. *)
```
GC$[t:\Omega^\tau,\ldots]$
$\quad \{y \mapsto (\textit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\textit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\lambda n.\lambda t.M_{o,r,y}(t), m_r)\}$
$\quad (h : M_{o,r,y}(t), k : \textit{gccont})$
$=$ let $h = \textit{copy}[t, 0, y, r, o](h)$ in
$\quad$ iffirst r then x, t' . *redirect*$[t, 0, t', y, r, o](x, h, k)$ else *GCtail*$[t, y, r, o](h, k)$

```
(* Code to execute at the end of GC when redirection is done. *)
```
GCtail$[t:\Omega^\tau,\ldots]$
$\quad \{y \mapsto (\textit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\textit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\textit{RefRgn}\ o\ r\ y\ m_r, m_r)\}$
$\quad (h : M_{o,r,o}(t), k : \textit{gccont})$
$=$ freergn y;
$\quad$ let $y =$ newrgn $(\lambda\_ : (\Omega^\tau \times \Omega^\tau).$ int) in
$\quad$ cast$[\ldots]\ r \mapsto (\lambda n.\lambda t.M_{o,r,y}(t))$;
$\quad$ cast$[\ldots]\ y \mapsto (\textit{TupRgn}\ o\ r\ y)$;
$\quad k[y](\text{cast}[\ldots : M_{o,r,o}(t) \triangleleft M_{o,r,y}(t)]\ h)$

```
(* Loop through the ref cells, redirecting ptrs from y to o. *)
(* h is the root pointer that we need to pass back to GCtail. *)
(* x is the boundary pointer that sweeps through r. *)
```
redirect$[t:\Omega^\tau, b$:Nat,$t' : \Omega^\tau,\ldots]$
$\quad \{y \mapsto (\textit{TupRgn}\ o\ r\ y, m_y), o \mapsto (\textit{TupRgn}\ o\ r\ o, m_o), r \mapsto (\textit{RefRgn}\ o\ r\ y\ b, m_r)\}$
$\quad (x : t$ at $r.b, h : M_{o,r,o}(t'), k : \textit{gccont})$
$=$ set $x \overset{M_{o,r,o}}{:=} \textit{copy}[t, b, y, r, o](\langle y,$ get $x\rangle)$;
$\quad$ cast$[\ldots]\ r \mapsto (\textit{RefRgn}\ o\ r\ y\ (b{+}1))$;
$\quad$ ifnext $x{+}1$ then x, t . *redirect*$[t, b{+}1, t', y, r, o](x, h, k)$ else *GCtail*$[t, y, r, o](h)$

Fig. 6. Generational GC

and use direct-style rather than continuation passing style where convenient, to make the code more readable. The code of the *copy* function is not shown and requires additions to our language which are outside the scope of this paper.

Since the types of regions themselves refer to those regions, there is a circularity problem when we create a region that prevents us from creating the region with its proper type. In *GCtail* we show how to work around this problem using cast: we first create Y with a dummy type, and then cast it to the desired type (the proof needs to show that the new type preserves the type of all objects in the region; this is trivial since the region is empty). The language could allow the type of a new region to refer to that new region, but that would still not help when creating several regions whose types refer to each other.

6 Related work

The calculus of capabilities [3] was the first calculus that tried to provide safe explicit memory deallocation while allowing dangling pointers. The linear handling of our regions was strongly influenced by that work. But they did not attempt to provide any form of strong update or to solve any of the other problems related to low-level memory management.

Alias types [17, 20] also have a lot in common. In that work, strong update is the only form of update available. Separating object initialization from allocation is very easy, as is explicit deallocation and memory reuse. But the calculus keeps the notion of location abstract, preventing any reasoning on the relative position of memory objects. Also the flexibility comes at the cost of requiring a static description of all the possible aliases of the object being assigned to. This information is often simply not available.

The Vault language [4] takes the work on alias types and tries to both extend it and give it a surface syntax (so as to enable to programmer to give that needed aliasing information). In the first paper, they mostly show how to integrate classical intuitionistic references with alias-types-style statically tracked references. They also show that tracking references to region objects allows an alias-type system to subsume a region type system. The main limitation compared to our work is that you have to choose once and for all whether a reference should be intuitionistic or linear (i.e. *tracked*).

In the second paper [5] they try to address that limitation by introducing the notions of *adoption* which allows the user to make a linear reference intuitionistic, and *focus* which does the converse. The implementation of *adoption* is unclear and seems somewhat incompatible with a low-level language. As for *focus*, it allows some of what we want (such as strong update in the presence of arbitrary aliasing), except for the fact that the restrictions are much too severe, especially the one that requires the type change to be limited to the scope of the construct.

In the work on Typed Assembly Language [12], the authors shows a simple way to handle the problem of separating allocation from object initialization, without resorting to any form of linearity, but this approach is inherently very limited and does not seem to lend itself to any other application.

Wang and Appel [21] proposed to build a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management. Their type system does not support existential packages, so they have to rely on a closure conversion algorithm that represents closures as datatypes [19, 15]. This makes closures transparent, making it easy for the copy function to analyze, but it

requires whole program analysis and has major drawbacks in the presence of separate compilation. More importantly, their algorithm does not address the problems linked to cycles or generations, and only presents an impractical treatment of forwarding pointers.

Monnier et al. [10] has shown how to use intensional type analysis [7] to solve the more serious problems of Wang and Appel's work and to extend it to deal with a very primitive form of generational collection, but which does not allow cycles or even any form of destructive assignment on the part of the mutator.

Shao et al. [16] proposed to use CiC as the type calculus of programming language so as to be able to manipulate explicit proofs. We reuse their idea with the same purpose of allowing sophisticated type manipulation and arbitrary cast operations.

7 Conclusion

We have presented a novel type system that offers an unusual flexibility to play with the typing of memory locations. This type system offers the ability to choose any mix of linear or intuitionistic typing of references and to change this choice over time to adapt it to the current needs. It is able to handle strong update of memory locations even in the presence of unknown aliasing patterns. The reliance on CiC allows very sophisticated type manipulations.

We have shown how to use it to solve the difficult problem of type checking a generational garbage collector, including proper handling of cycles and while allowing the mutator to perform destructive assignment on reference cells. We have also shown how to use this same type system to encode a stack of activation records inside a region.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, Jan. 1999.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Symposium on Programming Languages Design and Implementation*. ACM Press, May 2001.

[5] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Symposium on Programming Languages Design and Implementation*. ACM Press, May 2002.

[6] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[7] B. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, pages 130–141, Jan. 1995.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[9] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[10] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. Technical Report Yale/DCS/1205, Yale University, 2000.

[11] S. Monnier and Z. Shao. Typed regions. Technical Report Yale/DCS/1242, Yale University, 2002.

[12] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, Jan. 1998.

[13] G. C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*, Jan. 1997.

[14] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

[15] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, pages 717–740, 1972.

[16] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, Portland, OR, Jan. 2002.

[17] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, 2000.

[18] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, Jan. 1994.

[19] A. Tolmach and D. P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[20] D. Walker and G. Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, Aug. 2000.

[21] D. C. Wang and A. W. Appel. Safe garbage collection = regions + intensional type analysis. Technical Report TR-609-99, Princeton University, 1999.

[22] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Symposium on Principles of Programming Languages*, Jan. 2001.

[23] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

$$
\begin{array}{ll}
(\textit{sort}) & s ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{ptm}) & \varphi ::= s \mid x \mid \lambda x{:}\varphi.\,\varphi \mid \varphi\ \varphi \mid \Pi x{:}\varphi.\,\varphi \\
& \quad \mid \mathsf{Ind}(x{:}\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\,(i,\varphi) \mid \mathsf{Elim}[\varphi,\varphi](\varphi)\{\vec{\varphi}\}
\end{array}
$$

Fig. 7. Syntax of the calculus of inductive constructions.

A Calculus of constructions

The syntax of CiC is shown in Fig 7. The λ term corresponds to the abstraction of the λ-calculus, and the Π term is a dependent product type. When the bound variable does not occur in the body, the product type is usually abbreviated as $\varphi \to \varphi$. In the terminology of pure type systems, Kind, Kscm, and Ext are the sorts.

CiC, as its name implies, extends the calculus of constructions with inductive definitions. An inductive definition can be written in a syntax similar to that of ML datatypes. For example, the following introduces an inductive definition of polymorphic lists:

$$\mathsf{Ind}(\mathit{List}{:}\mathsf{Kind} \to \mathsf{Kind})\{\mathit{nil} : \Pi x.\mathit{List}\ x \mid \mathit{cons} : \Pi x.x \to \mathit{List}\ x \to \mathit{List}\ x\}$$

The logic also provides elimination constructs for inductive definitions, which combine case analysis with a fixpoint operation. Objects of an inductive type can thus be iterated over using these constructs. In order for the induction to be well-founded and for iterators to terminate, a few constraints are imposed on the shape of inductive definitions; most importantly, the defined type can only occur positively in the arguments of its constructors.The calculus of inductive constructions has been shown to be strongly normalizing [23], hence the corresponding logic is consistent.

In the remainder of this paper, we will use more familiar mathematical notation , rather than the strict definition of CiC syntax given in this section. For example, inductive definitions will be presented in BNF format. We will, however, retain the Π notation, which can generally be read as a universal quantifier.

B Typing rules

The typing rules for the terms of our language are given here in Fig. 9 and 10. Those rules use the two following auxiliary type functions:

$$\mathit{upd}\ \varphi\ n\ \varphi' = \lambda m.\lambda t.\mathsf{if}\ (m = n)\ (\varphi'\ t)\ (\varphi\ m\ t)$$

$$
\begin{array}{ll}
\mathit{size}(\sigma_1 \times \sigma_2) & \Rightarrow \mathit{size}(\sigma_1) + \mathit{size}(\sigma_2) \\
\mathit{size}(\exists t{:}\kappa.\sigma) & \Rightarrow \mathit{size}(\sigma) \\
\mathit{size}(\exists r \in \vec{\rho}.\sigma) & \Rightarrow \mathit{size}(\sigma) \\
\mathit{size}(\_) & \Rightarrow 1
\end{array}
$$

The main judgments are:

- $\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$ if v has type σ in the given environment.
- $\Psi; \Delta; \Theta; \Gamma \vdash e$ if e is well-formed under the given environment.

$$\boxed{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad \vdash \Psi \qquad \Delta \vdash \Theta \qquad \Psi;\Theta \vdash M}$$

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Delta \qquad \Delta \vdash t : \kappa}{\vdash \Delta, t{:}\kappa} \qquad \frac{\Delta \vdash \sigma_i : \Omega}{\Delta \vdash x_0{:}\sigma_0, \ldots, x_n{:}\sigma_n}$$

$$\frac{\bullet \vdash \varphi_{ij} : \kappa \qquad \bullet \vdash \nu_i : \mathsf{R}\ \kappa}{\vdash \nu_0.0 \mapsto \varphi_{00}, \ldots, \nu_m.n \mapsto \varphi_{mn}} \qquad \frac{\Delta \vdash \rho_i \mapsto \theta_i}{\Delta \vdash \rho_0 \mapsto \theta_0, \ldots, \rho_n \mapsto \theta_n}$$

$$\frac{\Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash \rho_i : \mathsf{R}\ \kappa}{\Delta \vdash \rho \mapsto \vec{\rho}} \qquad \frac{\Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash \varphi : \mathsf{Nat} \to \kappa \to \Omega}{\Delta \vdash \rho \mapsto (\varphi, n)}$$

$$\frac{\forall j{:}0..n_i{-}1\ .\ \Psi;\bullet;\bullet;\bullet \vdash M(\nu_i.j) : \varphi_i\ j\ (\Psi(\nu_i.j))}{\Psi;\nu_0 \mapsto (\varphi_0, n_0), \ldots, \nu_n \mapsto (\varphi_n, n_n) \vdash M} \qquad \frac{\Psi;\Theta \vdash M \qquad \Psi;\Theta;\bullet;\bullet \vdash e}{\vdash (M;\Theta;\Psi;e)}$$

Fig. 8. Environment formation rules.

- $\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \sigma$ if v points to an object of type σ.
- $\sigma \lhd \sigma'$ if σ is a subtype of σ'.
- $\Theta \vdash \rho \in \vec{\rho}$ if ρ is indeed one of $\vec{\rho}$.
- $\Theta \vdash \Theta'$ if Θ' can be used in lieu of Θ because it contains the same regions with the same types (but they can differ w.r.t. to region aliases).

The typing rules for environments are given in Fig. 8 together with the definition of a well-formed machine state $\vdash (M;\Theta;\Psi;e)$.

B.1 Properties of the language

We state here a few lemmas used in the proof of type soundness. The proofs can be found in the companion technical report [11]. Since our type language is CiC, we know it is strongly normalizing and confluent.

The proofs of the lemmas below do not present any unusual difficulty, contrary to what was the case in [10] where the widen operator introduced a lot of extra trouble. The substitution lemma for types turned out to be simpler than expected when substituting ν for r in a Θ such as $\{r \mapsto \nu, \nu \mapsto \ldots\}$.

One annoyance is that $\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma$ has to be re-proven each time Θ is changed. Luckily, reconstructing the new proof is simple since Θ is only used in those rules in order to verify that the witness of a package of type $\exists r \in \vec{\rho}.\sigma$ is indeed in $\vec{\rho}$.

Lemma 1 (Subsumption).
If $\sigma \lhd \sigma'$ and $\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma$ then $\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma'$.

Lemma 2 (Deallocation).
If $\Psi;\Delta;\Theta;\Gamma \vdash e$ and $\rho \notin Dom(\Theta)$ then $\Psi\backslash\rho;\Delta;\Theta;\Gamma \vdash e$.

Lemma 3 (Substitution).
If $\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma$ then:

- *If $\Psi;\Delta;\Theta;\Gamma\{x{:}\sigma\} \vdash v' : \sigma'$ then $\Psi;\Delta;\Theta;\Gamma \vdash v'[v/x] : \sigma'$.*

$\boxed{\Psi;\Delta;\Theta;\Gamma \vdash e}$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \qquad \bullet \vdash \vec{\sigma} : \overrightarrow{t:\kappa} \qquad \Theta \vdash \Theta'[\vec{\sigma}/\vec{t}] \qquad \Psi;\Delta;\Theta;\Gamma \vdash v_i : \sigma_i[\vec{\sigma}/\vec{t}]}{\Psi;\Delta;\Theta;\Gamma \vdash v[\vec{\sigma}](\vec{v})} \qquad \frac{\Psi;\Delta;\bullet;\Gamma \vdash v : \mathsf{int}}{\Psi;\Delta;\bullet;\Gamma \vdash \mathsf{halt}\ v}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \varphi\ n\ \tau \qquad \Psi;\Delta;\Theta,\rho \mapsto (\varphi, n{+}1);\Gamma, x{:}\tau\ \mathsf{at}\ \rho.n \vdash e}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{let}\ x = \mathsf{put}[\rho,\tau]\ v\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash op : \sigma \qquad \Psi;\Delta;\Theta;\Gamma, x{:}\sigma \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x{:}\sigma = op\ \mathsf{in}\ e} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash e}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{freergn}\ \rho; e}$$

$$\frac{\Psi;\Delta;\Theta,\rho \mapsto (\varphi', n);\Gamma \vdash e \qquad \Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash P : \Pi i{:}0..n{-}1.\, \Pi t{:}\kappa.\, (\varphi\ i\ t) \triangleleft (\varphi'\ i\ t)}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{cast}[P]\ \rho \mapsto \varphi'; e} \qquad \frac{\Delta \vdash \varphi : \mathsf{Nat} \to \kappa \to \Omega \qquad \Psi;\Delta, r{:}\mathsf{R}\ \kappa;\Theta, r \mapsto (\varphi, 0);\Gamma \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash v : \tau\ \mathsf{at}\ \rho.m \qquad \Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash v' : \varphi'\ \tau \qquad \mathit{size}(\varphi'\ \tau) = \mathit{size}(\varphi\ m\ \tau) \qquad \Psi;\Delta;\Theta,\rho \mapsto (\mathit{upd}\ \varphi\ m\ \varphi', n);\Gamma \vdash e}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{set}\ v \overset{\varphi'}{:=} v'; e}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \exists r' \in \vec{\rho}.\sigma \qquad \Delta \vdash \rho_i : \mathsf{R}\ \kappa \qquad \Psi;\Delta, r{:}\mathsf{R}\ \kappa;\Theta, r \mapsto \vec{\rho};\Gamma, x{:}\sigma[r/r'] \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ \langle r, x\rangle = \mathsf{openrgn}\ v\ \mathsf{in}\ e} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \exists t'{:}\kappa.\sigma \qquad \Psi;\Delta, t{:}\kappa;\Theta;\Gamma, x{:}\sigma[t/t'] \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e}$$

$$\frac{\Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Psi;\Delta;\Theta,\rho \mapsto (\varphi, 0);\Gamma \vdash e_2 \qquad \Psi;\Delta, t{:}\kappa;\Theta,\rho \mapsto (\varphi, n);\Gamma, x{:}t\ \mathsf{at}\ \rho.0 \vdash e_1}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{iffirst}\ \rho\ (x, t\ .\ e_1)\ (e_2)}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \sigma \qquad i = \mathit{size}\ \sigma \qquad \Psi;\Delta;\Theta;\Gamma \vdash v : \tau\ \mathsf{at}\ \rho.m \qquad \Psi;\Delta;\Theta,\rho \mapsto (\varphi, m{+}1);\Gamma \vdash e_2 \qquad \Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Psi;\Delta, t{:}\kappa;\Theta,\rho \mapsto (\varphi, n);\Gamma, x{:}t\ \mathsf{at}\ \rho.m{+}1 \vdash e_1}{\Psi;\Delta;\Theta,\rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{ifnext}\ v{+}i\ (x, t\ .\ e_1)\ (e_2)}$$

Fig. 9. Static semantics of the language.

– *If* $\Psi;\Delta;\Theta;\Gamma\{x{:}\sigma\} \vdash e$ *then* $\Psi;\Delta;\Theta;\Gamma \vdash e[v/x]$.

If $\Delta \vdash \varphi : \kappa$ *then:*

– *If* $\Psi;\Delta\{t{:}\kappa\};\Theta;\Gamma \vdash v : \sigma$ *then* $\Psi;\Delta[\varphi/t];\Theta[\varphi/t];\Gamma[\varphi/t] \vdash v[\varphi/t] : \sigma[\varphi/t]$.
– *If* $\Psi;\Delta\{t{:}\kappa\};\Theta;\Gamma \vdash e$ *then* $\Psi;\Delta[\varphi/t];\Theta[\varphi/t];\Gamma[\varphi/t] \vdash e[\varphi/t]$.

Lemma 4 (Type Preservation).
If $\vdash (M;\Theta;\Psi;e)$ *and* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$ *then* $\vdash (M';\Theta';\Psi';e')$.

$\boxed{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma}$

$$\frac{}{\Psi;\Delta;\Theta;\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Psi;\Delta;\Theta;\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v_i : \sigma_i}{\Psi;\Delta;\Theta;\Gamma \vdash (v_1, v_2) : \sigma_1 \times \sigma_2}$$

$$\frac{\nu.n \in \mathit{Dom}(\Psi) \Rightarrow \tau = \Psi(\nu.n)}{\Psi;\Delta;\Theta;\Gamma \vdash \nu.n : \tau \ \mathsf{at}\ \nu.n} \qquad \frac{\Psi;\Delta;\Theta;\overrightarrow{x:\sigma} \vdash e}{\Psi;\Delta';\Theta';\Gamma \vdash \lambda[\Delta]\{\Theta\}(\overrightarrow{x:\sigma}).e : \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0}$$

$$\frac{\Delta \vdash \varphi : \kappa \qquad \Psi;\Delta;\Theta;\Gamma \vdash v : \sigma[\varphi/t]}{\Psi;\Delta;\Theta;\Gamma \vdash \langle \varphi, v\rangle : \exists t{:}\kappa.\sigma} \qquad \frac{\begin{array}{c}\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma[\rho/r] \\ \Delta \vdash \rho : \mathsf{R}\ \kappa \qquad \Delta \vdash \rho_i : \mathsf{R}\ \kappa \qquad \Theta \vdash \rho \in \vec{\rho}\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \langle \rho, v\rangle : \exists r \in \vec{\rho}.\sigma}$$

$\boxed{\Theta \vdash \rho \in \vec{\rho} \qquad \Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \sigma \qquad \Psi;\Delta;\Theta;\Gamma \vdash op : \sigma}$

$$\frac{\rho \in \vec{\rho}}{\Theta \vdash \rho \in \vec{\rho}} \qquad \frac{\Theta \vdash \rho_i \in \vec{\rho'}}{\Theta, \rho \mapsto \vec{\rho} \vdash \rho \in \vec{\rho'}} \qquad \frac{\Theta(\rho) = (\varphi, m)}{\Theta \vdash \tau \ \mathsf{at}\ \rho.n \mapsto \varphi\ n\ \tau} \qquad \frac{\Theta \vdash \tau \ \mathsf{at}\ \rho_i.n \mapsto \sigma[\rho_i/\rho]}{\Theta, \rho \mapsto \vec{\rho} \vdash \tau \ \mathsf{at}\ \rho.n \mapsto \sigma}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \tau \ \mathsf{at}\ \rho.n \qquad \Theta \vdash \tau \ \mathsf{at}\ \rho.n \mapsto \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \sigma} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v \mapsto \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{get}\ v : \sigma}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi;\Delta;\Theta;\Gamma \vdash \pi_i v : \sigma_i} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash v : \sigma' \qquad \Delta \vdash P : \sigma' \lhd \sigma}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{cast}[P]\ v : \sigma}$$

$\boxed{\Theta \vdash^{\mathsf{live}} \rho \qquad \Delta \vdash \vec{\sigma} : \vec{t{:}\kappa} \qquad \Theta \vdash \Theta' \qquad \sigma \lhd \sigma'}$

$$\frac{}{\Theta, \rho \mapsto (\varphi, n) \vdash^{\mathsf{live}} \rho} \qquad \frac{\Theta \vdash^{\mathsf{live}} \rho_i}{\Theta, \rho \mapsto \vec{\rho} \vdash^{\mathsf{live}} \rho} \qquad \frac{\Delta \vdash \sigma : \kappa \qquad \Delta \vdash \vec{\sigma} : \overrightarrow{t{:}\kappa}[\sigma/t]}{\Delta \vdash \sigma, \vec{\sigma} : t{:}\kappa, \overrightarrow{t{:}\kappa}}$$

$$\frac{}{\Theta \vdash \Theta} \qquad \frac{\Theta \vdash \Theta'}{\Theta, \rho \mapsto (\varphi, n) \vdash \Theta', \rho \mapsto (\varphi, n)} \qquad \frac{\Theta \vdash \Theta' \qquad \Theta \vdash \rho \in \vec{\rho'}}{\Theta \vdash \Theta', \rho \mapsto \vec{\rho'}} \qquad \frac{\Theta \vdash \Theta'}{\Theta, \rho \mapsto \vec{\rho'} \vdash \Theta'}$$

$$\frac{}{\sigma \lhd \sigma} \qquad \frac{\sigma_1 \lhd \sigma_1' \qquad \sigma_2 \lhd \sigma_2'}{\sigma_1 \times \sigma_2 \lhd \sigma_1' \times \sigma_2'} \qquad \frac{\vec{\rho} \subset \vec{\rho'} \qquad \sigma \lhd \sigma'}{\exists r \in \vec{\rho}.\sigma \lhd \exists r \in \vec{\rho'}.\sigma'}$$

Fig. 10. Typing values and operations, with auxiliary rules.

Lemma 5 (Progress).
If $\vdash (M;\Theta;\Psi;e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a state* $(M';\Theta';\Psi';e')$ *such that* $(M;\Theta;\Psi;e) \Longrightarrow (M';\Theta';\Psi';e')$.

Lemma 6 (Complete Collection).
If $\vdash (M;\Theta;\Psi;e)$ *and* $(M;\Theta;\Psi;e) \Longrightarrow^* (M';\Theta';\Psi';e')$
and $\forall \nu.n \in \mathit{Dom}(M)\ .\ \nu \in \mathit{Dom}(\Theta)$ *then* $\forall \nu.n \in \mathit{Dom}(M')\ .\ \nu \in \mathit{Dom}(\Theta')$.
In particular, if $e' = \mathsf{halt}\ v$ *then* $M' = \bullet$.